Published in OOPSLA'94 Conference Proceedings, Portland, OR, October 1994.

Sifting Out the Gold

Delivering Compact Applications from an Exploratory Object-Oriented Programming Environment

Ole Agesen
Computer Science Department
Stanford University
Stanford, CA 94305, USA
agesen@cs.stanford.edu

David Ungar
Sun Microsystems Laboratories
2550 Garcia Ave
Mountain View, CA 94043, USA
David.Ungar@sun.com

Abstract

Integrated, dynamically-typed object-oriented programming environments offer many advantages, but have trouble producing small, self-contained applications. Recent advances in type inference have made it possible to build an application extractor for Self. The extractor was able to extract a medium-sized application in a few minutes. The extracted application runs in a tenth the space of the original environment. Except for extracting reflection and sends with computed selectors, the extractor runs without human intervention and fully preserves the behavior of the application.

1 Introduction

Why doesn't everyone use a language like Smalltalk? After all, object-oriented programming is the rage, and the Smalltalk-80 and -V programming environments pamper the programmer with good tools, a comfortable GUI, clean semantics, and rapid turnaround. It may be that programmers are unfamiliar with or do not like the language. On the other hand, more pragmatic reasons may be influencing them:

- Smalltalk's speed lags an order of magnitude behind C's;
- Smalltalk's lack of static type checking can mask errors;
- Smalltalk's integrated development environment and lack of static typing make it hard to deliver small applications.

Recently, great strides have been made towards leveling the playing field for performance and error detection of dynamically-typed object-oriented languages [Deutsch & Schiffman 1984, Hölzle & Ungar 1994, Agesen et al. 1993]. Application size, however, has remained an obstacle preventing programmers from using dynamically-typed exploratory programming environments. Without some sort of application extraction, even the simplest program such as "Hello World" has the same size as the entire programming environment.

This paper reports on the results of applying type inference to the task of extracting applications from Self [Ungar & Smith 1987], a streamlined successor of Smalltalk. We have succeeded in extracting several benchmarks and a real application in regular use with only minor modifications. The image size of the extracted application is one tenth that of the complete Self system.

1.1 Contributions

We have designed, implemented, and tested a new algorithm to extract applications from exploratory object-oriented programming environments. This algorithm is

- automatic: it runs without programmer intervention, although it will accept advice to improve the results;
- sound: it preserves the full semantics of the extracted application;
- efficient: extraction, including type inference, takes minutes;
- effective: extracted applications shrink by an order of magnitude (of course depending on how big the application is in the first place; we describe a specific medium-sized example in detail below);
- general: although our implementation is specific to Self, the algorithm can be readily applied to, say, Smalltalk-80 and CLOS.

2 Background

Since we describe our extraction algorithm as it applies to Self, we briefly introduce Self in Section 2.1. The example application we use to describe the extraction algorithm is presented in Section 2.2.

2.1 Overview of Self

An object in Self consists of slots. A slot has a name and a value. Slot names are always strings, but slot values can be any Self object, either a data object or a method object. Slot names can be marked with an asterisk to show that they designate a parent. When sending a message, if a match does not occur within the receiving object, its parents' slots are searched, and then slots in the parents' parents, and so on. In Self, any object can potentially be a parent for any number of children, or a child of any object. When an object is found in a slot as a result of a message send, it is run; no extra step is needed to invoke a method. A data object without code runs by just returning itself.

Self might be called an extremely object-oriented language, because what other languages do with special mechanisms, Self does with objects and messages. The uniformity extends to control flow structures (Self implements these with dynamic dispatching), variable scoping and accessing mechanisms (Self allows inheritance of state), and primitive types and operations (even an integer is an object in Self). This total devotion to the object-message paradigm simplifies the language, making it a convenient test-bed for application extraction.

2.2 Our Sample Application: PrimitiveMaker

PrimitiveMaker, the application we refer to throughout this paper, was written before our work on extraction began. The extractor was nevertheless able to extract PrimitiveMaker unmodified, although we had to make two minor changes elsewhere in the Self world for extraction to succeed (see Section 4).

While detailed knowledge of PrimitiveMaker is not needed, an overview is helpful. PrimitiveMaker is a preprocessor that generates Self and C glue code for foreign functions. For example, suppose a Self programmer needs to call the C function `open`. The programmer would create a "template file" containing a line naming the function and specifying the C types of its arguments and result. He would next run the PrimitiveMaker on the file in order to create three files. The first file contains a Self routine that calls the primitive. The second file contains a C stub that translates Self objects to C values, calls the `open` routine, and translates the return value back to a Self object. The third file creates an entry in the Self virtual machine's primitive dispatch table. Most of the 600 primitives in the Self system are implemented in this manner, using PrimitiveMaker.

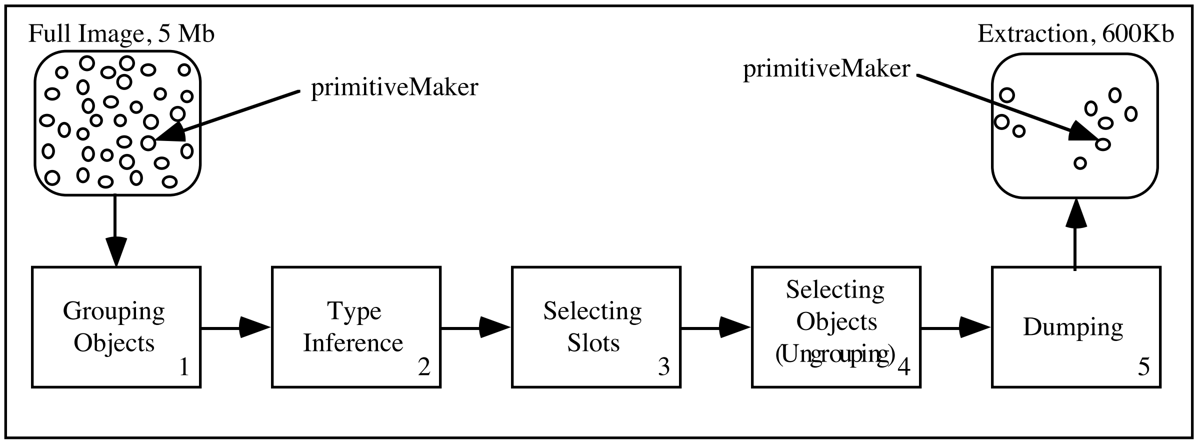


Figure 1. The extraction algorithm takes five steps to extract an application.

3 How Application Extraction Works

The input to the extractor is an image of objects containing an application to be extracted. Typically, the image is the "standard Self image" consisting of core objects, glue, user interface, and additionally one or more applications. The standard image alone comprises 5.5 Mbytes. An *application* is anything that can be executed by sending a single message to an object. We designate the object and invoked method the "main object" and "main method," respectively. Like the `main` function in a C program, they explicitly define how to execute the application. In addition, they implicitly delineate the application: certain objects must be present for the execution to succeed, whereas other objects can be safely omitted. When extracting PrimitiveMaker, the main object is `primitiveMaker` and the main method is the one that compiles a template file.

Our extraction algorithm consists of five steps, as shown in Figure 1. The harder sub-problem of extraction is identifying what needs to be extracted. The first four steps accomplish this, the fifth step simply harvesting the fruit and dumping the selected objects. The following sections describe each step in detail.

3.1 Step 1: Grouping Objects

Type inference, the second step, is expensive. So expensive, in fact, that it is infeasible to infer types for each object individually. In Smalltalk, to overcome this problem, one would infer types for classes instead of individual objects, effectively analyzing all instances of a class in parallel. In Self, there is no such concept as classes built into the language. Instead, we *group* objects, allowing the type inference step to analyze groups of objects rather than individual objects; see Figure 2. For example, all smallInteger objects form a group. Other groups are bigInts, points, and true (this group has only one member). In our implementation, groups correspond roughly to clone families (grouping is actually based on a structural comparison since in Self there is no way to tell if two objects belong to the same clone family or just happen to be similar).

Grouping involves a trade-off: finer groups make type inference more expensive (there are more groups to analyze), whereas coarser groups make type inference less precise (it is not possible to distinguish the types of different members of a group). Ideally, we should group objects according to their behavior (types), but this is not possible without knowing the types. Instead, we approximate behavior with structure. For extraction, this seems to yield a satisfactory compromise between precision and performance of type inference. In general, an

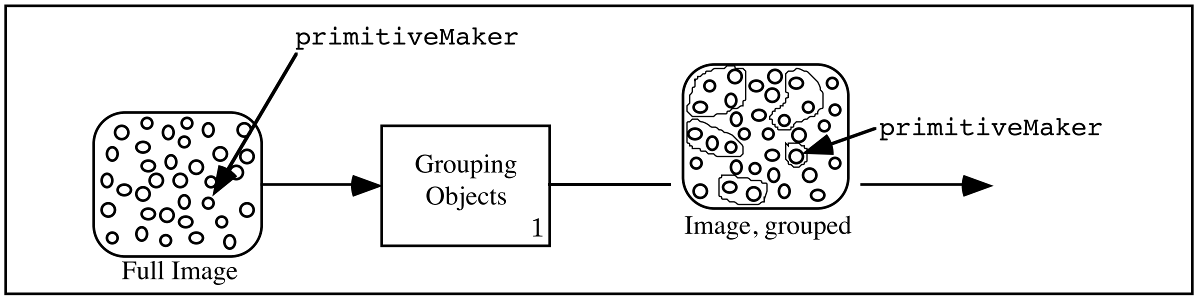


Figure 2. Step 1 groups the objects in the image according to their structure.

adaptive approach to grouping may be necessary, analogous to the adaptive type inference algorithms described in [Plevyak & Chien 1993, Agesen 1994].

The extractor only groups objects as it encounters them. This procrastination saves the effort of grouping objects that are not reached during extraction, and most objects are never grouped: extracting PrimitiveMaker from an image with 100,000 objects, only 5000 of these are grouped, into 600 different groups. Sometimes it pays to be lazy.

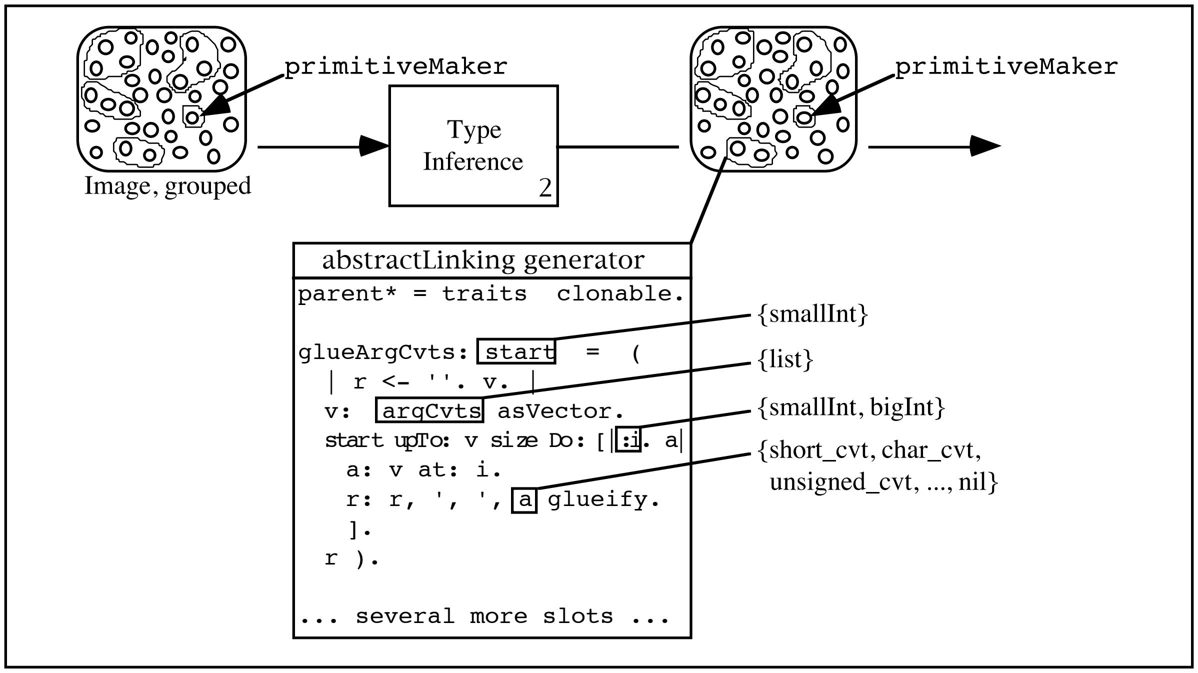
3.2 Step 2: Type Inference

Suppose we need to extract an application that sends the message `size` to the result of some expression R. The image in which the application has been developed may contain hundreds of methods called `size`, for rectangles, dictionaries, sets, arrays and other kinds of objects. Which of the myriad `size` methods should be extracted?

The extractor needs some information about the kinds of objects that could possibly result from computing R. For example, if it (somehow) could discover that R never returns hash tables, there would be no need to extract the size method for hash tables (unless, of course, it was needed elsewhere). In a prior publication, we report on a *type inference* algorithm that can compute this kind of information [Agesen et al. 1993]. (See also [Plevyak & Chien 1993] for more recent results, and [Agesen 1994] for a comparison of different variations of the type inference algorithm). The second step in the extraction of PrimitiveMaker invokes our type inference algorithm on PrimitiveMaker[1]. After type inference, every expression and slot in PrimitiveMaker is annotated with the set of groups it might possible yield. Henceforth we shall refer to such a set as a *type*.

Figure 3 shows the types inferred for the `glueArgCvts:` method found in a certain `generator` group in PrimitiveMaker. To avoid clutter only selected type annotations are shown. Without any further explanation of the role of this method, let us take a look at the annotations computed by the inferencer and see what we can learn. For example, the type of the `i` argument is the set {smallInt, bigInt}. This annotation means that during execution of PrimitiveMaker, the value of `i` will always be an object which is a member of either the smallInt or bigInt group. Likewise, the type annotation for the expression `argCvts` is {list}. Anticipating the next step, the reader can now see that only the `asVector` method that applies to lists should be extracted (unless of course `asVector` is sent to another group elsewhere in the application). Finally, the type of `a` is a set of some 20 groups; thus the `glueify` send is highly polymorphic. This send alone tells us to extract the 20 respective `glueify` methods.

[1]Invoking the type inference algorithm on PrimitiveMaker does not mean that PrimitiveMaker has already been isolated from the rest of the image. It merely means that we invoke the type inference algorithm on the main method and object of PrimitiveMaker.



```
parent* = traits  clonable.

glueArgCvts: start  =  (
  | r <- ''. v. |
  v: argCvts asVector.
  start upTo: v size Do: [| :i. a|
    a: v at: i.
    r: r, ', ', a glueify.
    ].
  r ).

... several more slots ...
```

Figure 3. Step 2 invokes the type inference algorithm on the main object and method, producing type annotations for all methods that the application may execute.

The inferred types are *sound*, but *conservative estimates* of the dynamic (run-time) types. "Sound" means that the types of the actual objects will always be included in the inferred types. For example, since the type of `i` was inferred to be {smallInt, bigInt}, soundness means that at any moment during any execution of PrimitiveMaker, the object stored in the `i` slot will either be a smallInt or a bigInt. "Conservative" means that the inferred types may overestimate the dynamic types. This explains why nil shows up in the type of `a` (if `nil` ever was to occur at run time, PrimitiveMaker would hit an error, since `nil` does not understand `glueify`). Conservatism is not a "weakness" that is unique to our algorithm: any practical type inference algorithm must be conservative, since exact type inference is uncomputable.

The soundness of the type inferencer allows our extraction algorithm to guarantee full preservation of behavior of the extracted application. The fact that the inferred types are conservative means that we may sometimes extract objects that will actually not be needed. We have not yet been able to measure the degree to which this happens, but we are considering applying a coverage tool to the extracted application to measure how much of it is used dynamically over several typical runs.

Type inference is the most expensive step. It takes almost 3 minutes of CPU time on a 50 MHz SPARC-station 10 to infer types for PrimitiveMaker (this actually includes the time spent grouping, but grouping is negligible). It also consumes a fair amount of memory. Approximately 10 Mbytes of data structures are built.

3.3 Step 3: Selecting Slots

An application extractor that altered the behavior of the applications it extracted would not be very useful, so in this step the extractor identifies a set of slots that is large enough to fully preserve the behavior of the application, yet as small as the available type information permits. We call such a set of slots "small and sufficient." See Figure 4 for an illustration. A small and sufficient set of slots can be computed by simulating each send in the application, so we attack two sub-problems: collecting sends and simulating sends. (An efficient

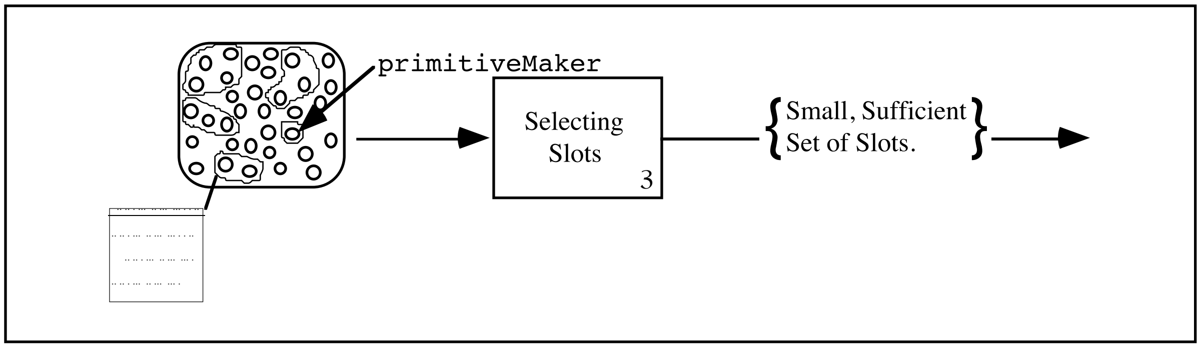


Figure 4. Step 3 selects a set of slots to extract. The set is large enough to preserve the behavior of the application, but as small as the available type information permit.

implementation will probably interleave the two sub-problems; here we keep them separate for clarity).

Collecting sends. A transitive closure operation is used to collect a sound (and small) approximation to the syntactic sends that the application being extracted may execute. First, initialize a set with the sends in the main method. Then, for each send in the set, type information leads to the set of methods that may be invoked. The sends in these methods are added to the set and the induction step is repeated until the set of possible sends grows no more.

The procedure shown in Figure 5 implements this transitive closure. When invoked on the main method, it returns a list of all sends that the application being extracted may execute.

Simulating sends. We now use the sends to *mark* a small sufficient set of slots. The idea is to simulate the sends one by one. During the simulation of a send, the extractor marks all slots whose presence is needed to preserve the behavior of the send. Specifically, given a send and a possible receiver, it marks the target slots that the send may invoke. In addition, if a target slot is not found directly in the receiver but instead is inherited through a chain of parent links, it marks the parent slots on the inheritance chain leading to the target slot. No further slots are marked. Since the marked slots preserves the behavior of every send for every possible receiver, and since in Self all computation is performed by passing messages, the behavior of the application as a whole is preserved.

The procedure, `MarkMinSuffSlots`, shown in Figure 6, formalizes this way of marking a sufficient set of slots. The set of slots marked is minimal in the restricted sense that no slot can be safely omitted if everything that the type information predicts may happen during execution can in fact happen. In other words: the set of slots marked is as small as the available type information permits. Of course, availability of more accurate type information would allow the extractor to mark a smaller set of slots and still know that it is sufficient (as long as the type information remains sound, a sufficient set of slots is marked).

The only complication that may arise is specific to Self: a lookup may encounter dynamic inheritance. In that case, the extractor cannot straightforwardly apply the standard lookup algorithm. Instead, to be sound, it must search through *all* possible dynamic parents. To do this, it consults the type information that was computed for the dynamic parent slot, which gives a — usually small — set of possible parents (groups, actually) that the lookup proceeds into.

Given the type information, collecting sends and marking slots takes relatively little time. For PrimitiveMaker the combined CPU time for `CollectSends` and `MarkMinSuffSlots` is approximately 20 seconds. A total of 2232 slots are marked.

The result in this third step of extraction is a mapping, MarkedSlots, from groups to sets of slots:

```
procedure CollectSends(m: method)
   var sends: list;
begin
   "mark m";   (* To break cycles when processing recursive methods. *)
   for each send in m.code do
      sends.append(send);
      for each receiverGroup in type(send.receiverExp) do
         accessedSlot := receiverGroup.lookup(send.selector);
         if accessedSlot.isMethod and "it is unmarked" then
            sends.append(CollectSends(accessedSlot.contents));
         end;
      end;
   end;
   return sends;
end CollectSends;
```

Figure 5. Recursive procedure for collecting the set of all sends that an application may execute.

```
procedure MarkMinSuffSlots(APP)
begin
   "clear all slot marks";
   for each send in CollectSends(APP) do
      for each receiverGroup in type(send.receiverExp) do
         "simulate the lookup of send.selector starting in receiverGroup";
         for each matchingSlot that the lookup found do
            "set the mark in matchingSlot";
            "set the mark in each parent slot on the path from the
             receiverGroup to the group containing the matchingSlot";
         end;
      end;
   end;
end MarkMinSuffSlots;
```

Figure 6. Procedure to mark a sufficient set of slots for an application, given the sends that may be executed.

for a group g, MarkedSlots(g) is the slots in g that were marked by `MarkMinSuffSlots`.

3.4 Step 4: Selecting Objects (Ungrouping)

The fourth step, illustrated in Figure 7, determines for each individual object in the image whether or not it needs to be extracted. This is done by carefully "lowering" the group level result from the previous step to the object level.

Let x be an object and let g = group(x). If MarkedSlots(g) = Ø, then there is no reason to extract x, since no slot in x will ever be accessed by the application. Instead, the extracted version can safely replace any reference to x by a reference to the empty object. Seen in this light, step 3 finds those groups from which one or more members must be extracted. A naive way to proceed would be to extract all the members from the groups which had one or more slots marked, and extract no members from the remaining groups. Succinctly:

$$\text{ExtractSet}' = \{x \in \text{Image} \mid \text{MarkedSlots}(\text{group}(x)) \neq \emptyset\}.$$

This extraction is clearly sound but potentially much larger than need be. For example, just because the application needs *some* point object, it is unlikely that *every* point object in the image will be needed.

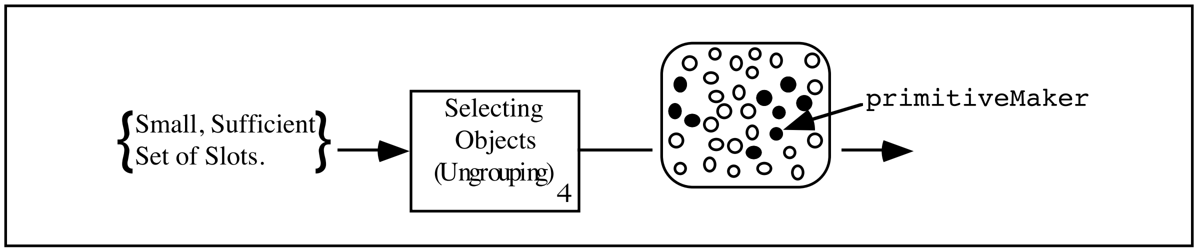

Small, Sufficient Set of Slots.
Selecting Objects (Ungrouping) 4
primitiveMaker

Figure 7. Step 4 maps the group-level set of slots from the previous step back onto the object-level.

The key to extracting a smaller but still sufficient set of objects is to view the problem of identifying the required objects as a reachability problem: if there is no way for the application to reach a specific object, there is no reason to extract it. This viewpoint leads to a smaller extraction, because the application is limited to accessing objects through the slots that were marked in Step 3.

Definition. Let APP be an application. An object is *APP-reachable* if and only if:

- it is the main object, or
- it is contained in a slot of an APP-reachable object x and the slot is in MarkedSlots(group(x)).

  (It may of course be the case that APP will never access a specific slot in the specific object x, but since we can not rule out the possibility, we declare the contents APP-reachable).

It is an easy task to convert these rules into code so we will omit it here. Applying the rules identifies a sufficient set of objects:

ExtractSet = {x∈Image | x is APP-reachable}.

Identifying the 457 objects that are PrimitiveMaker-reachable takes only 3 seconds, given the MarkedSlots mapping. These objects contain a total of 2272 marked slots. Comparing this with the 2232 group level slots in MarkedSlots we conclude that most groups have only a single reachable member. (Literal objects, i.e. smallIntegers, floats, and strings are excluded from ExtractSet, since they do not need to be extracted; instead they are implicitly created by the virtual machine at startup time).

3.5 Step 5: Dumping the Objects

The final step, illustrated in Figure 8, is to write out a representation of the objects in the ExtractSet that was identified in the previous step. Writing out a possibly circular structure of objects is a standard problem that has been previously addressed, e.g. by the Smalltalk-80 BOSS system for storing objects in a binary format [ParcPlace 1992, Section 27]. In our case, there is only a slight twist to the problem: when writing out an object x, only the slots given by MarkedSlots(group(x)) are written out.

The extractor writes a single source file. It is "stand-alone," i.e. can be read into an empty virtual machine. Subsequently, a binary image can be easily obtained by simply invoking the primitive that writes out an image.

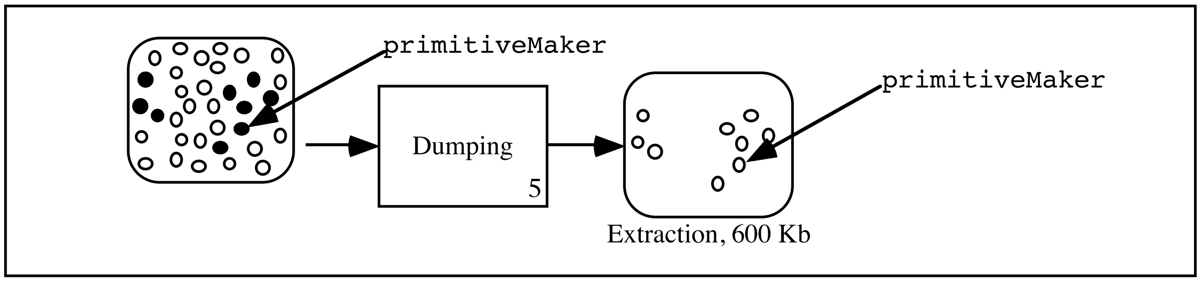


Figure 8. Step 5, finally, dumps the slots (and objects) that were selected by the previous four steps.

	Grouping	Type inference	Selecting slots	Selecting obj's	Dumping objects
Time	3 minutes		20 seconds	3 seconds	5 seconds
Output	10 Mbytes of type annotations		2232 slots	2272 slots in 457 objects	155Kb/5230 lines source (603 Kbytes image)

Table 1. Summary of the extraction process when applied to PrimitiveMaker. The starting point is a 5.5 Mb image containing primitiveMaker; the final result is a 603 Kb image, also containing PrimitiveMaker.

The extractor's dump files are quite naive. For example, even when similar objects could be obtained from each other by cloning, all of them are still constructed from scratch. The consequence is that merely looking at the size of the dump file can be misleading. Comparing image sizes gives a more consistent measure, although it should also be remarked that Self images tend to be larger than Smalltalk images. For PrimitiveMaker, a total of 155 Kb or 5230 lines of source was dumped in 5 seconds. The corresponding image size is 603 Kbytes, as compared to almost 6 Mbytes for the original image.

3.6 Summary of the Extraction Process

The extraction time is dominated by the type inference time. However, since the inferencer is incremental it would take less time to re-infer types after a change. At four minutes, the total extraction time seems not unreasonable for a relatively infrequent activity. Appendix A contains data for several more applications we have extracted. These include the full *Stanford Integer Benchmarks* suite, originally collected by John Hennessy and later used to characterize the run-time performance of the Self system [Chambers & Ungar 1991]. The appendix also reports on extracting *richards*, an operating system simulation benchmark, and *deltablue*, a multiway constraint solver algorithm [Sannella et al. 1993].

Table 1 summarizes the results of extracting PrimitiveMaker. For each step, the time required and the result produced are listed. For example, the dumping step executes in 5 seconds and produces 5230 lines of stand-alone Self source code (which can be converted to a 603 Kb image).

There are two road blocks that currently prevent us from measuring the performance of the extractor on many applications. First, the Self virtual machine defines some 600 primitives, but the type inference step only supports 200 of these. Second, the Self system contains a scheduler that implements concurrent processes; we have not yet generalized the extractor to cope with concurrency.

4 Discussion

Application extraction raises a number of issues, some specific to Self, some specific to our particular extractor, and others more general. Since we expect more people to use extraction than to implement it (someday!), we will first, in Section 4.1, delve into the way that a user's programming style interacts with extraction. Then, in Section 4.2, we discuss the issues that impact the design of the extractor.

4.1 Programming Style Issues

Should you be worried about extraction while you are writing programs? Are some styles of programming more conducive to extraction than others? The Self system, having evolved for years without regard to extraction, is a good case to study. Our recent experience suggests that although extraction can cope with most stylistic variations, two idioms, sends with computed selectors and reflection, can pose problems.

4.1.1 Computed Selectors (Performs)

The `_Perform` primitive sends messages whose names cannot be statically determined. This lack of information forces the type inferencer to treat it very

```
self _Perform: (tokenList removeFirst, ':') With: true.
```

```
flag: tokenList removeFirst.  "Get name of flag to set."
flag = 'canFail'        ifTrue: [self canFail: true].
flag = 'canAWS'         ifTrue: [self canAWS: true].
flag = 'passFailHandle' ifTrue: [self passFailHandle: true].
```

Figure 9. A perform found in PrimitiveMaker (above the line) and equivalent code not using perform (below the line).

conservatively; thus potentially forcing extraction of objects that are not really needed.

The top half of Figure 9 shows a perform from PrimitiveMaker. The performed selector is computed by removing a string from a list and appending a colon to it. The receiver of the perform is `self` (i.e., `primitiveMaker`) and there is a single argument, `true`. It turns out — but the extractor currently can not determine this — that there are only three possibilities for the performed selector, so the code shown in the bottom half of Figure 9 is equivalent. Not knowing the performed selector, except that the syntax reveals that it passes one argument, the extractor must assume that *any* 1-argument method in `primitiveMaker` can be invoked. There are 46 such methods all of which must then be extracted. But then the ball starts rolling and any method or object accessed from any of the 46 methods also need to be extracted, and so on.

We considered modifying PrimitiveMaker to not use perform, but decided that it was more general to let the extractor accept advice from the programmer. In the specific example, he would specify that the selector performed is one of `canFail:`, `canAWS:`, and `passFailHandle:`. With this advice, precise type inference is again possible. (In many cases, the uncertainty of the performed selector is of minor significance; then the programmer does not have to give any advice).

If the programmer (inadvertently) gives unsound advice, the extracted application might not run. Fortunately, it is possible to detect if an unsound advice was used during extraction. The idea is to incorporate the advice into the extracted application together with code verifying its soundness. In the above example, the extracted application would check that the performed selector is one of the three which the programmer specified. Should a fourth selector occur, it is of course too late to repair the damage, but at least it is possible to output a precise error message.

An alternative to advice-taking is to improve the type inference algorithm, possibly by extending it to track values. In the present example it would not have worked anyway, since the performed selector is read in from a file.

4.1.2 Reflection

Normally, the only thing that can be done with an object is to send it a message and observe the result. Sometimes, though, it is necessary to inquire about the structure of an object. For example, the user interface needs to find out the names of an object's slots in order to display it. This manipulation of the structure of an object, known as *structural reflection,* is accomplished in Self via meta-objects called *mirrors*. While working on PrimitiveMaker, the extractor bumped up against two uses of reflection, although neither of them were specific to PrimitiveMaker. They were both found in objects implementing standard data types that are used by most applications.

Reflection first crept into PrimitiveMaker through the `printString` method in collections. PrimitiveMaker prints out list objects during its execution. Lists generate their printString by invoking a general method that applies to several kinds of collections. This general method reflects upon the elements of the collection to see if they implement `printString` and if not, to look them up in a cache of path names for all prototypes. The consequence of this reflection was dire: since our type inference algorithm does no range analysis, it had to assume that any of the objects in the path cache might be pulled out. This in turn forced extraction of every prototype in the system! It was a one line change to avoid this use of reflection, but it did incur the cost of making the collection `printString` method less robust: a collection with unprintable members is now itself unprintable.

The second way in which PrimitiveMaker uses reflection is related to the previously discussed performs. When a perform send fails, e.g. with "message not understood," a signal is generated by the virtual machine. The signal takes the form of a message sent to the currently executing process object. When the process object receives the error signal it calls into the reflective domain to handle the signal. While we think it is within reach to extract some reflective code, our type inference algorithm is currently not able to deal with this specific example without running out of memory.

The aggravating circumstance about both cases described above is that neither was part of the application proper. Rather, they were both found in library code that most applications, not just PrimitiveMaker, would end up using. This failure of extraction could severely hinder a person somewhat unfamiliar with Self who tried to extract an application that, say, generated printStrings for sets. How would he feel about getting *every* object in the path cache extracted as part of his application? Would he understand it? Probably not. Would he find it acceptable? Certainly not. We see no easy solution to this problem, except rewriting existing code on a case by case basis as it is deemed necessary, and encouraging programmers writing new code to keep the extraction technology in mind. Thus far, we have encountered only few and easily handled problems.

4.2 Issues in the Design of an Extractor

We discuss three aspects that the designer of an extraction algorithm should be aware of. Section 4.2.1 considers resources required for extraction. Section 4.2.2 discusses how much behavior should be preserved across extraction. Section 4.2.3 discusses the granularity that extraction should be based on.

4.2.1 Resources Required to Extract an Application

How good is our extractor? Several criteria could apply. Some are obvious, like speed of the extraction process and memory consumption. All other things being equal, a faster, less memory demanding algorithm is preferable, but it should also be noted that extraction is not like compilation or debugging. The latter activities are performed repeatedly during program development. Extraction, on the other hand, can conceivably be limited to taking place just before delivery, so the resources consumed are less important. Taking a few minutes, our extractor seems to be fast enough to be useful.

Another resource to consider is programmer time. Depending on how powerful the extraction algorithm is, more or less programmer involvement may be required. This issue should not be played down. It has potentially serious consequences, e.g. for reuse: if the programmer knows that he may have to guide the extractor through code he is about to reuse, he may choose not to reuse, because the effort required to obtain a sufficient understanding of the reused code may exceed the savings from reusing. Although our type inference algorithm still has room for improvement, it seems to be adequate to drive an extractor and only rarely require programmer intervention.

4.2.2 How Much Behavior is Preserved Across Extraction?

Although an extractor should produce the most compact applications it can, an algorithm that extracts the smallest amount of code may or may not be the best choice. While less code extracted is better, the unavoidable consequence may be that less behavior is preserved. One can distinguish between three different levels of behavior preservation. In order of increasing quality they are:

- *Correct programs only*. "If the unextracted program executes without error, it is guaranteed that the extracted program will execute without error. No further guarantees apply." This is the minimally acceptable guarantee to provide, but even so this level is dangerous. For example, if there *is* a bug in the application (and isn't there always?), all bets are off when running the extracted application because error conditions may go unnoticed and the application silently produces erroneous output. In some sense this minimal degree of behavior preservation is similar to switching off array bounds checking before shipping an application written in, say, Modula-2, a routine practice in our industry. This level may be acceptable when the consequences of unanticipated behavior are small relative to the likelihood of encountering undiscovered bugs.
- *Some error*. "If the unextracted program encounters an error, it is guaranteed that the extracted program will also encounter some error, but this may be a different error happening later in the execution." Compared with the previous level of behavior preservation, this level has the advantage that extraction will not convert a visible error into a stealthy error. The person running the application will be notified that an error has happened, but no further promises are given. In an extreme case, the application would run for long enough in the erroneous state to output erroneous results.
- *First error*. "If the unextracted program encounters an error, the extracted program will encounter exactly the same error." This level of behavior preservation facilitates debugging of errors encountered after delivery (the most expensive kind of errors). It also has the advantage that the extracted application will behave, to the highest possible degree, exactly as the unextracted application, enabling extrapolation of both successful and failing test runs from the unextracted application to the extracted application. For these reasons, we have crafted our extractor to operate at this level.

The reason that our algorithm attains the strongest guarantee is that it extracts enough code to preserve the behavior of every send in the extracted application, including the behavior of sends that may fail. We have not yet been able to quantify how much smaller a typical extracted application could be if we were willing to settle with one of the two weaker guarantees. In our case, to measure this, we would have had to modify our type inference algorithm to exploit the increased freedom; this, however, was beyond the initial scope of our work which takes the type inference algorithm for given. The trade-offs between different levels of behavior preservation vs. amount of code (vs. time to extract etc.) merit future exploration.

4.2.3 Granularity

The smallest unit that can be extracted is an important consideration because the larger it is, the more excess baggage may be dragged along when extracting the parts that were deemed necessary to extract. We identify three different granularities that extraction can be based on. From coarser to finer they are:

- *Module-based*. A module-based extractor includes or excludes modules as a whole from the extracted application. A module is a language (and programming style) dependent feature, but typically consists of a set of objects and classes. C++, while not employing extraction, uses a module-based approach for including code. For example, if a C++ programmer wants to use a class, he does this by including the module containing the class. The module is typically a ".o-file," and it may contain several other classes and/or objects, all of which will end up in his application.

- *Object/class-based.* An object-based extractor extracts objects (or classes) as a whole. Objects and classes are typically smaller than modules and so less excess baggage will be extracted along with the necessary objects. For example, ParcPlace Smalltalk implements manual class-based extraction: the programmer can specify a list of classes to be removed from the image [ParcPlace 1992, Section 16].
- *Slot/attribute-based.* A slot-based extractor such as our algorithm offers an even finer resolution because it filters out unused methods and variables.

Other granularities are possible, including still finer ones. It may, for instance, be reasonable to eliminate dead code within methods. We have not yet quantified the difference between the above three granularities with respect to the amount extracted, but we plan to do so by first computing a set of sufficient slots. Then we can "round off" to whole objects and dump an object-based extraction, and round off to modules and dump a module-based extraction. The object- and module-based extractions obtained in this manner should be very good since rounding off is done only at the very end, after having delineated using the finer slot-based resolution.

5 Previous Work

Although type inference is not the focus of the present paper, it does provide the foundation for our extraction algorithm. One of the earliest reports on type inference is [Borning & Ingalls 1981]. They describe a combined type declaration and inference system for Smalltalk and in the conclusion they write — with great foresight:

> Also, the type system may help in tracing control flow, thus making it possible to produce application modules containing just the code needed to run a particular application.

The type system that we have applied is different from their early scheme. Without the precise type information produced by our type inference algorithm, automatic extraction would have been much harder, if not impossible.

ParcPlace Smalltalk, *ObjectWorks Release 4.1*, provides guidance on "deploying an application" [ParcPlace 1992, Section 16]. An interactive tool, the Stripper, can be used to remove classes from the system. The Stripper has built-in knowledge of which classes specifically support program development (e.g., the compiler classes) and it can remove these classes from the image. The programmer may specify a list of extra classes to remove. There is no guarantees, however, that an essential class is not removed. Smalltalk/V contains a tool with similar functionality, called the "Cloner." Smalltalk/V also has the "Object Library Builder." When given a set of root objects and an "import list," the Builder extracts all objects reachable by transitive closure from the roots, up to, but not including, objects on the import list. Typically, the import list consists of standard classes and objects. The Builder does not trace control flow, but computes a simple transitive closure, hence may extract unused objects [Digitalk 1993].

The Lucid Common Lisp system included Treeshake, a delivery toolkit module that attempted to extract applications by decompiling them and discovering the interconnections between modules [Boreczky & Rowe 1993]. It made no attempts to resolve generic functions according to the types that would be used at run time; all arms of any generic function used would be included. In the end, it was not considered an unqualified success [White 1994].

Allegro Common Lisp used a different strategy for application delivery: instead of extracting an application, it could start an application running in an empty world and lazily load in modules as needed [Boreczky & Rowe 1993, White 1994]. Although this technique can reduce the memory footprint, it would seem to still require as much disk space as the full system. Furthermore, it may increase startup time of the application.

Palsberg and Schwartzbach describe an algorithm for eliminating dead code [Palsberg & Schwartzbach

1993]. Like our algorithm, the core of their algorithm is a type inference system that enables conservative control flow analysis. The environments are different, however. Our algorithm analyzes an image, sifting through objects (and code) to determine what can be safely omitted. Their algorithm analyzes a textual representation of a program to find code that will never be executed. Another difference is that their algorithm works on programs written in BOPL (Basic Object Programming Language), a minimal object-based language designed for teaching and studying programming language issues. While BOPL contains the essentials of any "real" object-oriented language, the minimality of the language and implementation makes it unsuitable for writing large programs. Thus, Palsberg and Schwartzbach did not have the luxury of a large body of existing code to test their algorithm on.

Statically-typed object-oriented languages such as Beta, C++, and Eiffel typically rely on "traditional" delivery tools inherited from procedural languages. Usually, modules are compiled into ".o-files," possibly combining these into static or dynamic library files. Applications are then built by invoking a linker such as `ld` under Unix. The linker starts with a specified set of .o-files and computes the transitive closure of all references in these files and any referenced .o-files in libraries. Then all reachable .o-files are concatenated to form the application.

This traditional approach differs from the Self/Smalltalk situation in that there is no initial confusion between the programming environment and the application. While it is not necessary to carve the application out of a massive development environment (it is just built from the parts that programmer or compiler specifies) unreachable code may still enter the final application — and increasingly so, as programs and libraries get larger and code reuse becomes more common. According to a recent study by Srivastava who analyzed several large C++ programs, such programs contain significantly more unreachable code than programs written in procedural languages like C and Fortran. Using a simple static analysis algorithm he found that up to 26% of the code in large C++ programs is unreachable [Srivastava 1992]. We believe that by applying our algorithm, an even larger amount of code can effectively be proven dead and thus safely omitted from the linked applications.

6 Conclusions

The problem of delivering a modestly-sized application without the overhead of a large development environment has hindered the acceptance of object-oriented exploratory programming environments like Smalltalk-80, Self, and CLOS. Recent advances in the technology of type inference have made it possible to construct an automated application extractor, and we have done so for Self.

To be useful, extraction must preserve the behavior of the extracted application. We identified three different levels of behavior preservation. Our extractor provides the strongest guarantees: even in the presence of bugs in the application, extraction will not alter the behavior.

We identified three granularities that extraction can be based on: modules, objects, or slots. Our extractor operates at the finest granularity, that of slots, in order to winnow out as much unneeded data as possible.

Although more experience is needed, the early results are promising: a real application, and several benchmarks, have been extracted in a few minutes of CPU time. The resulting objects take only one tenth the space of the full development environment.

Our extractor operates without human intervention. However, two programming idioms can confound it: sends with computed selectors (performs) and reflection. Currently, programmer intervention may be called upon in such cases.

In the future, we plan to apply our extractor to more applications and to generalize it to better cope with performs and reflection.

Acknowledgments. Discussions with several persons helped improve this paper. In particular, we would like to thank L. Bak, B. W. Chang, L. P. Deutsch, U. Hölzle, O. L. Madsen, J. Palsberg, M. Sakkinen, and M. I. Schwartzbach. B. Vokach-Brodsky and A. Wirfs-Brock helped us understand Smalltalk/V. We also thank the anonymous reviewers for their helpful feedback.

The first author was generously supported by The Natural Sciences Faculty of Aarhus University, The Danish Research Academy, and Sun Microsystems Laboratories. The Self project was supported by National Science Foundation Presidential Young Investigator Grant #CCR-8657631, Sun Microsystems, IBM, Apple Computer, Cray Laboratories, Tandem Computers, NCR, Texas Instruments, and DEC.

References

[Agesen 1994] Agesen, O. Constraint-Based Type Inference and Parametric Polymorphism. To be presented at *International Static Analysis Symposium (SAS'94),* Namur, Belgium, September 28-30, 1994.

[Agesen et al. 1993] Agesen, O., Palsberg, J., Schwartzbach, M. I. Type Inference of Self: Analysis of Objects with Dynamic and Multiple Inheritance. In *Proc. ECOOP '93, Seventh European Conference on Object-Oriented Programming,* pages 247-267. Springer-Verlag (LNCS 707), July 1993.

[Boreczky & Rowe 1993] Boreczky, J., Rowe, L., Building Common Lisp Applications with Reasonable Performance. In *Proc. Lisp Users and Vendors Conference*, 1993. Reprinted in SIGPLAN Lisp Pointers, 6(3).

[Borning & Ingalls 1982] Borning, A. H., Ingalls, D. H. H. A Type Declaration and Inference System for Smalltalk. In *Ninth Symposium on Principles of Programming Languages,* pages 133-141. ACM Press, January 1982.

[Chambers & Ungar 1991] Chambers, C., Ungar, D. Making Pure Object-Oriented Languages Practical. In *Proc. OOPSLA '91, ACM SIGPLAN Sixth Annual Conference on Object-Oriented Programming Systems, Languages and Applications,* pages 1-15, October 1991.

[Deutsch & Schiffman 1984] Deutsch, L. P., Schiffman, A. Efficient Implementation of the Smalltalk-80 System. In *Proceedings of the 11th Symposium on the Principles of Programming Languages*, Salt Lake City, UT, 1984.

[Digitalk 1993] *Smalltalk/V for Win32 Programming, Reference Manual,* Chapter 17: "Object Libraries and Library Builder," Digitalk Inc., 1993.

[Hölzle & Ungar 1994] Hölzle, U., Ungar, D. Optimizing Dynamically-Dispatched Calls with Run-Time Type Feedback. In *Proceedings of the ACM SIGPLAN '94 Conference on Programming Language Design and Implementation (PLDI)*, pages 326-336, Orlando, FL, June 1994.

[Palsberg & Schwartzbach 1993] Palsberg, J., Schwartzbach, M. I. *Object-Oriented Type Systems*. John Wiley & Sons, 1993.

[ParcPlace 1992] *ObjectWorks Smalltalk User's Guide*. Release 4.1, Section 16: "Deploying an Application," Section 28: "Binary Object Streaming Service,". ParcPlace Systems, 1992.

[Plevyak & Chien 1993] Plevyak, J., Chien, A. A. *Incremental Inference of Concrete Types*, Technical Report R-93-1829, Department of Computer Science, University of Illinois Urbana-Champaign, 1993.

[Sanella et al. 1993] Sannella, M., Maloney, J., Freeman-Benson, B., Borning, A. Multi-way versus One-way Constraints in User Interfaces: Experience with the DeltaBlue Algorithm. *Software - Practice and Experience* 23 (5), pages 529-566, 1993.

[Srivastava 1992] Srivastava, A. Unreachable Procedures in Object-Oriented Programming. *ACM Letters on Programming Languages and Systems* 1 (4) (Dec. 1992), pages 355-364.

[Ungar & Smith 1987] Ungar, D., Smith, R. Self: The Power of Simplicity. In *Proc. OOPSLA '87, Object-Oriented Programming Systems, Languages and Applications,* pages 227-241, 1987. Also published in Lisp and Symbolic Computation 4(3), Kluwer Academic Publishers, June, 1991.

[White 1994] White, J. L., Private conversation, February, 1994.

	Type inf.	Extraction	#Objects	#Slots	Dumped Src	Image
PrimMaker	3 min	28 sec	457	2272	155 Kb	603 Kb
deltablue	3 min	34 sec	69	455	59 Kb	366 Kb
richards	13 sec	24 sec	71	453	59 Kb	357 Kb
perm	16 sec	9 sec	58	336	48 Kb	312 Kb
towers	6 sec	9 sec	56	352	48 Kb	315 Kb
queens	9 sec	8 sec	54	340	48 Kb	315 Kb
matrix mult	17 sec	10 sec	56	356	49 Kb	338 Kb
puzzle	89 sec	16 sec	57	469	63 Kb	428 Kb
quicksort	29 sec	11 sec	55	382	51 Kb	354 Kb
bubblesort	5 sec	11 sec	56	378	50 Kb	332 Kb
treesort	15 sec	11 sec	58	392	52 Kb	361 Kb
stanford-8	3 sec	21 sec	78	605	75 Kb	521 Kb

Table 2. Summary of the extraction of PrimMaker, deltablue, richards, and the Stanford Integer Benchmarks.

Appendix A

We have extracted several applications besides PrimitiveMaker: *deltablue*, a multiway constraint solver [Sannella et al. 1993], *richards*, an operating system simulation, the eight *Stanford Integer Benchmarks* individually (perm, towers, queens, matrix multiply, puzzle, quicksort, bubblesort, treesort), and the same eight combined into a single application (stanford-8). The original intention with the integer benchmarks was to characterize the run-time performance of Self [Chambers & Ungar 1991]. Our algorithm extracted all these applications without requiring any advice or modifications to the benchmarks or the system as a whole.

Table 2 summarizes the results of extracting PrimitiveMaker and the eleven benchmarks from the standard Self image. The time measurements deserve some explanation. First, the type inference times are showing a wide variation. This is mostly due to our type inference algorithm being incremental: subsequent executions of it can benefit from results of previous executions. So, while it took 3 minutes to infer types for deltablue, this work is helping in all the subsequent benchmarks which are processed in a matter of seconds typically.

Most of the extracted applications have similar sizes. The reason is that the majority of the code extracted is not in the application proper, but rather is “library code” such as integers, strings, and vectors, which they have in common. This also explains why the sum of the sizes of the individual benchmarks is 2755 Kb whereas the size of stanford-8 is only 521 Kb.

A significant body of code that all the applications force extraction of is the bigInt implementation. To quantify this we repeated all the measurements, extracting from an image without bigInts. The result showed a fairly consistent drop in extracted image size of roughly 100 Kbytes.

But why are bigInts extracted in the first place when all the benchmarks work without them? The answer is found in the type inference step. It does no range analysis, hence must assume that integer arithmetic can overflow. BigInts are then extracted as a result of analyzing the Self code that handles arithmetic overflow (the code coerces smallInts to bigInts and retries the operation). This is an area we hope to improve: either attempting to automatically discover when bigInts can safely be omitted by performing some amount of range analysis, or alternatively, making it easy for the programmer to request that bigInts are not extracted.